Form with function

The task before us — Turn the abstract to concrete – Functions are at hand

Agenda

- **Start function-based chrestomathics**

Here we go

- Question: Why bother with functions?
  - Makes problem solving possible. Without them you would need to figure out
    - How to communicate with your keyboard to read what it's being typed
    - How to communicate to your display so that information can be displayed
    - How to …
- Question: Why bother with non-standard functions?
  - Tasks important to you can be solved and reused as wanted
    - Module url.py lets us skip tedious web processing setup to do something interesting

You are *already very* experienced with functions

- Program wc.py – uses modules, built-in functions, and string manipulation methods

```
import url
reply = input( 'Enter web file link: ' )
link = reply.strip()
contents = url.get_contents( link )
nbr_lines = contents.count( '\n' )
words = contents.split()
nbr_words = len( words )
nbc_chars = len( contents )
print( 'nl =', nbr_lines )
print( 'nw =', nbr_words )
print( 'nc =', nbc_chars )
```

-

Changing our *modus operandi*

- Past problem solving
  - The solution was a single program file
- Future problem solving
  - The solution will involve multiple files
    - One file will still be a program that orchestrates the problem solving
    - The other files will be *modules* (*libraries*) that define *functions* to support the problem solving
- *Functions provide an easier, clearer, and reusable way for the manipulation and calculation of information*

Module harb.py and program inger.py

```
''' Purpose: introduce modules
'''

def add( a, b ) :
    ''' Returns value of a and b
    '''

    result = a + b
    return result

def negate( x ) :
    ''' Returns inverse of x
    '''

    result = -x
    return result
```

```
''' Purpose: try out functions
'''
import harb

n1, n2 =  3, 14
n3, n4 = 15, 92
n5, n6 = -6, 53

t1 = harb.add( n1, n2 )
t2 = harb.add( n3, n4 )

print( t1, t2 )

i1 = harb.negate( n5 )
i2 = harb.negate( n6 )

print( i1, i2 )
```

Modules

- A *module* is a *named collection of functions*
  - Module name required to be a legal identifier
  - The functions of a module are stored in a Python file (.py) that matches the module name
    - All modules we develop will be *kept* in your CS 1112 class folder
  - Access to a module capabilities is via *importing*

    import *module_name*

Module harb.py and program inger.py

```python
''' Purpose: introduce modules
'''

def add( a, b ) :
    ''' Returns value of a and b
    '''

    result = a + b
    return result

def negate( x ) :
    ''' Returns inverse of x
    '''

    result = -x
    return result
```

```python
''' Purpose: try out functions
'''
import harb

n1, n2 =  3, 14
n3, n4 = 15, 92
n5, n6 = -6, 53

t1 = harb.add( n1, n2 )
t2 = harb.add( n3, n4 )

print( t1, t2 )

i1 = harb.negate( n5 )
i2 = harb.negate( n6 )

print( i1, i2 )
```

Functions

- A *function* is a *named block of code* to carry out some specific task
  - Function name required to be a legal identifier (just like variables are)
  - Functions can have special variables called *parameters* to store start up values
  - Access to a module capabilities is via *invocation*

    module_name . function_name (arguments)

    Arguments are start up values *copied/passed* to the function for initializing parameters
  - Functions can produce information that is handed back via a *return statement*

    return *return expression*
  - For functions without return returns Python supplies the value None

Module harb.py and program inger.py

```
''' Purpose: introduce modules
'''

def add( a, b ) :
    ''' Returns value of a and b
    '''

    result = a + b
    return result

def negate( x ) :
    ''' Returns inverse of x
    '''

    result = -x
    return result
```

```
''' Purpose: try out functions
'''
import harb

n1, n2 =  3, 14
n3, n4 = 15, 92
n5, n6 = -6, 53

t1 = harb.add( n1, n2 )
t2 = harb.add( n3, n4 )

print( t1, t2 )

i1 = harb.negate( n5 )
i2 = harb.negate( n6 )

print( i1, i2 )
```

Function definition syntax

A function definition looks like the following

```
def function_name( parameters ) :
      ''' header_comment
      '''
      action
```

- The keyword def signals that a function is being defined
- The *function_name* is an identifier that names the task
- The (*parameters*) is parenthesized list of variable names (identifiers)
- The *function_name* along with the *parameters* form the *function header*
- The colon : is a separator of the function header from the function and its *action*
- The header comment (*docstring*) describes the function
- The *action* is a *non-empty block of statements* that run when the function is invoked

Module harb.py and program inger.py

```
''' Purpose: introduce modules
'''

def add( a, b ) :
    ''' Returns value of a and b
    '''

    result = a + b
    return result

def negate( x ) :
    ''' Returns inverse of x
    '''

    result = -x
    return result
```

```
''' Purpose: try out functions
'''
import harb

n1, n2 =  3, 14
n3, n4 = 15, 92
n5, n6 = -6, 53

t1 = harb.add( n1, n2 )
t2 = harb.add( n3, n4 )

print( t1, t2 )

i1 = harb.negate( n5 )
i2 = harb.negate( n6 )

print( i1, i2 )
```

What happens with a function invocation

1. Checks whether the number of arguments and parameters match
   - No match: an execution error occurs
   - Match: arguments are *evaluated* (values are determined)
2. Sets aside some computer memory for carrying out the function
   - That memory is called an *activation record* or a *frame*
     - Stores values of the parameter variables
       - The parameter variables are initialized with the argument evaluations
     - Stores the values of other variables needed to perform its task

Module harb.py and program inger.py

```
''' Purpose: introduce modules
'''

def add( a, b ) :
    ''' Returns value of a and b
    '''

    result = a + b
    return result

def negate( x ) :
    ''' Returns inverse of x
    '''

    result = -x
    return result
```

```
''' Purpose: try out functions
'''
import harb

n1, n2 =  3, 14
n3, n4 = 15, 92
n5, n6 = -6, 53

t1 = harb.add( n1, n2 )
t2 = harb.add( n3, n4 )

print( t1, t2 )

i1 = harb.negate( n5 )
i2 = harb.negate( n6 )

print( i1, i2 )
```

What happens with a function invocation

3. The function's code now runs; i.e., it has the *flow of control*
   - When the function completes, the flow of control goes *back* to the code that did the invocation, and it is that code now continues being executed

Module harb.py and program inger.py

```python
''' Purpose: introduce modules
'''

def add( a, b ) :
    ''' Returns value of a and b
    '''

    result = a + b
    return result

def negate( x ) :
    ''' Returns inverse of x
    '''

    result = -x
    return result
```

```python
''' Purpose: try out functions
'''
import harb

n1, n2 =  3, 14
n3, n4 = 15, 92
n5, n6 = -6, 53

t1 = harb.add( n1, n2 )
t2 = harb.add( n3, n4 )

print( t1, t2 )

i1 = harb.negate( n5 )
i2 = harb.negate( n6 )

print( i1, i2 )
```

What happens when Python encounters a function invocation

4. If a return statement is reached, the *return value* is determined
   - If there is an expression following keyword return, its value is the *return value*
   - If there is no expression following keyword return, the *return value* is None

   If the code complete without reaching a return statement, the *return value* is set to None

   When the *return value* is set, the *execution of the function stops* and
   - The flow of control goes back to the statement that did the invocation
   - The *value of the invocation – its evaluation* – is the *return value*

Understand difference between parameter and argument

```
''' Purpose: introduce modules
'''

def add( a, b ) :
    ''' Returns value of a and b
    '''

    result = a + b
    return result

def negate( x ) :
    ''' Returns inverse of x
    '''

    result = -x
    return result
```

```
''' Purpose: try out functions
'''
import harb

n1, n2 =  3, 14
n3, n4 = 15, 92
n5, n6 = -6, 53

t1 = harb.add( n1, n2 )
t2 = harb.add( n3, n4 )

print( t1, t2 )

i1 = harb.negate( n5 )
i2 = harb.negate( n6 )

print( i1, i2 )
```

A taxonomy

The two main kinds of functions are

- Functions that have parameters and return values
- Functions that have parameters and have None as the return value

There can also be

- Functions that do not have parameters, but do return values
- Functions that do not have parameters and have None as the return value

Functions

- When designing a function we need to
  - Determine what information is needed upon startup
  - Determine what information is to be produced
  - Provide an algorithm for going from one to the other